# Distributed Systems

Chun-Feng Liao

廖峻鋒

Department of Computer Science

National Chengchi University

## Distributed Systems

# Remoting

Chun-Feng Liao

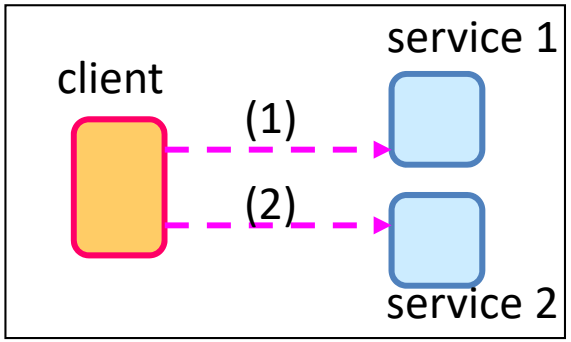
廖峻鋒

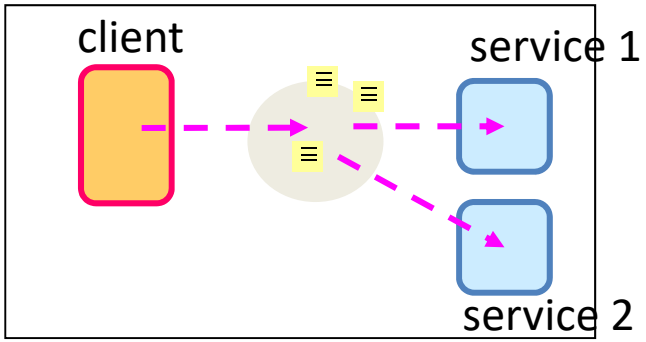
Dept. of Computer Science

National Chengchi University

# Basic Remoting Styles

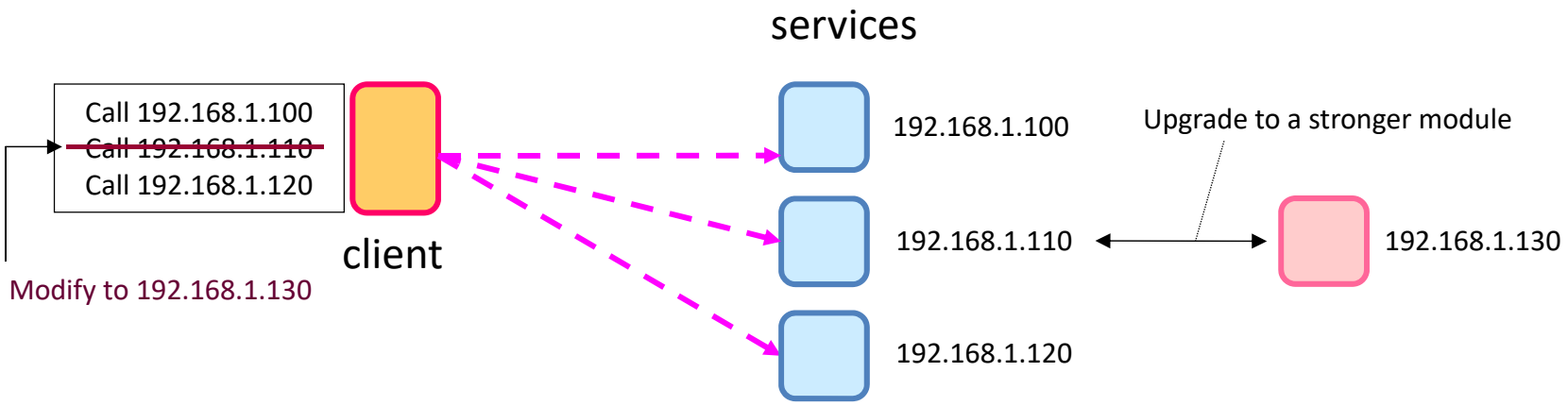
- Two major styles
  - Direct communication (remoting)
    - Java RMI; CORBA; Microsoft DCOM; .NET Remoting; SOAP
  - Indirect communication (messaging)
    - Kafka; RabitMQ; MQTT; Shared DB



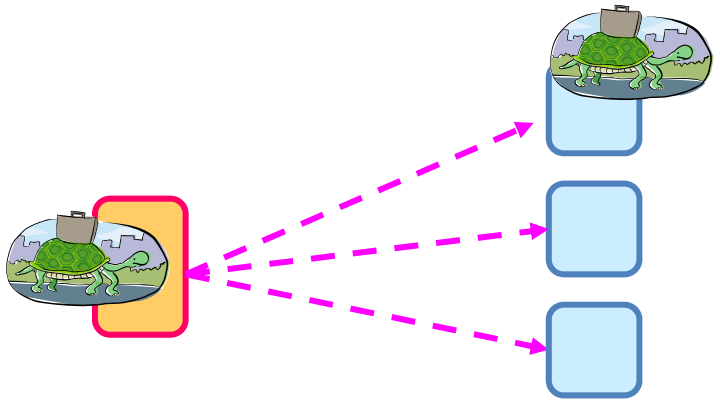
# Direct Communication

- Benefits
  - Simplified application development
- Drawbacks
  - Tightly coupled  on space (reference) and time (synchronous)
  - Fragile and hard to recover



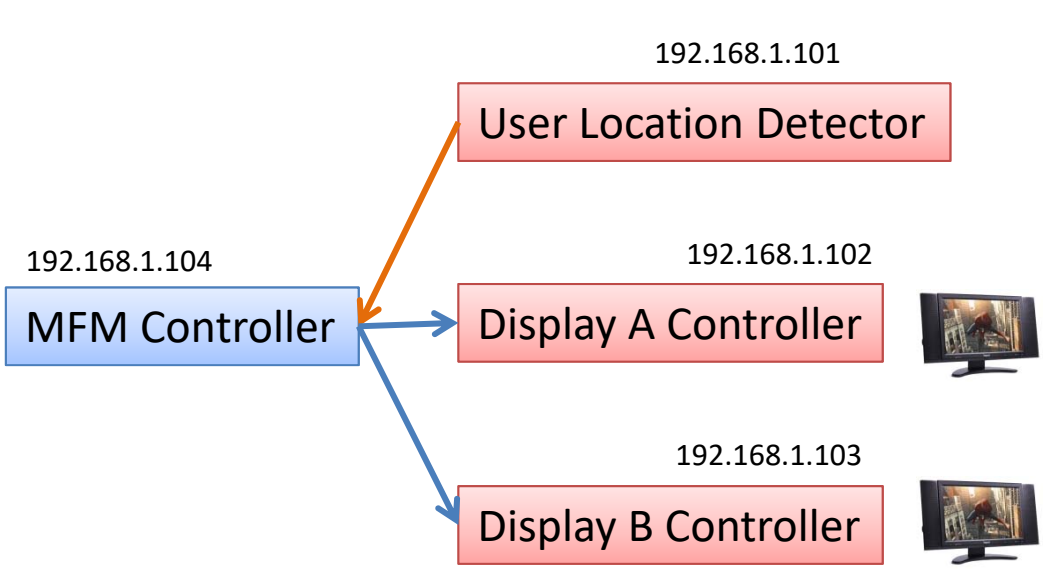
# Direct Communication

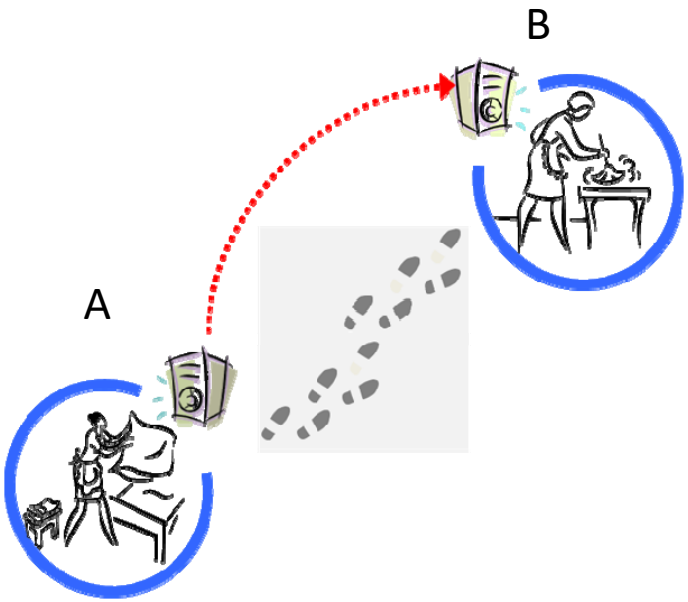
- Benefits
  - Simplified application development
- Drawbacks
  - Tightly coupled  on space (reference) and time (synchronous)
  - Fragile and hard to recover



# Direct Communication Example

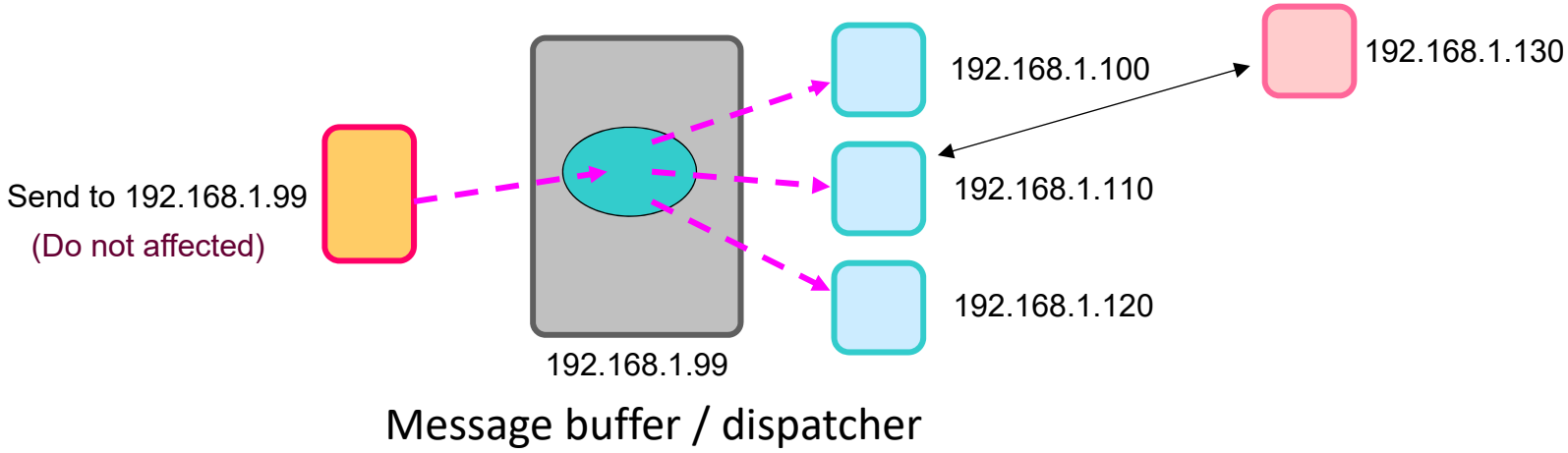
- Approach
  - A “main” process invokes remote procedures sequentially
  - Usually realized as RPC (Remote Procedure Call)



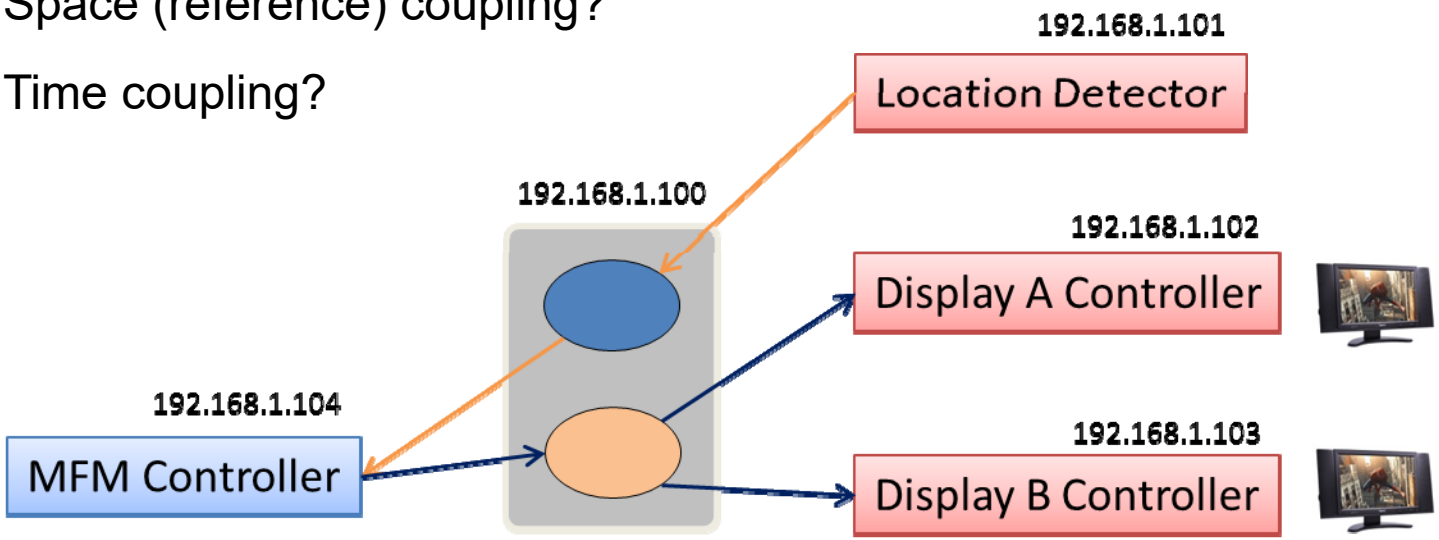
# Indirect Communication

- Will be detailed in the next lecture
- Benefits
  - Decoupling in both space and time
- Drawbacks
  - Single point of failure can be alleviated by clustering
  - Address binding of the broker can be alleviated by broker discovery
  - Application logic is de-centrialized



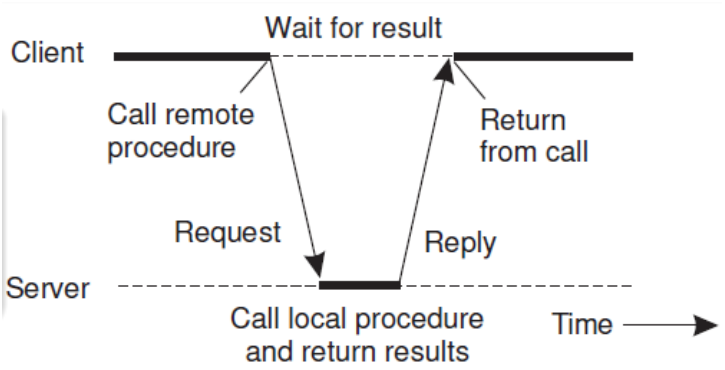
# Indirect Communication

- Approach
  - Introducing a centralized message buffer
  - Publish-subscribe driven
  - Coupling
    - Space (reference) coupling?
    - Time coupling?



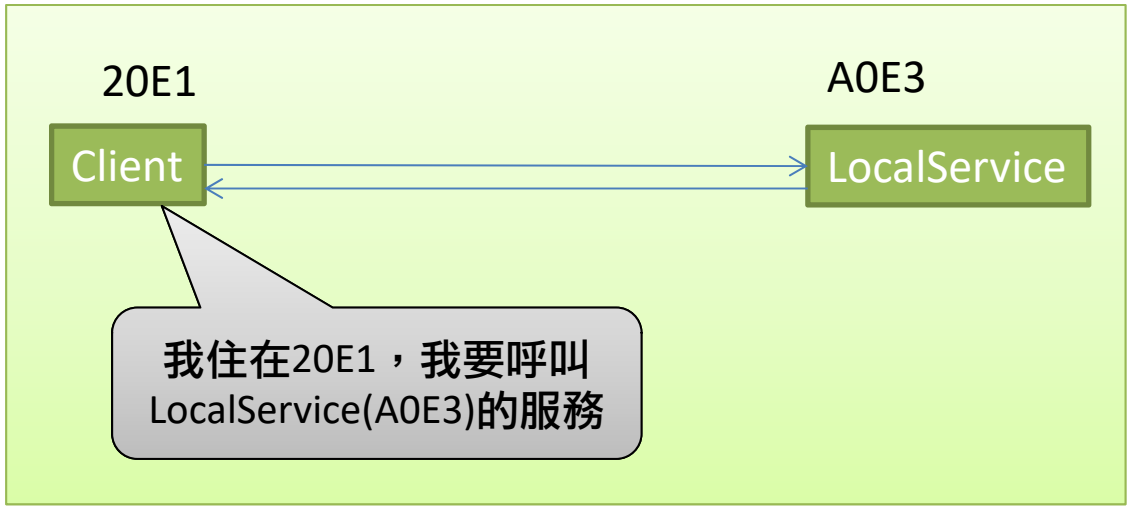
# Remote Procedure Call (RPC)

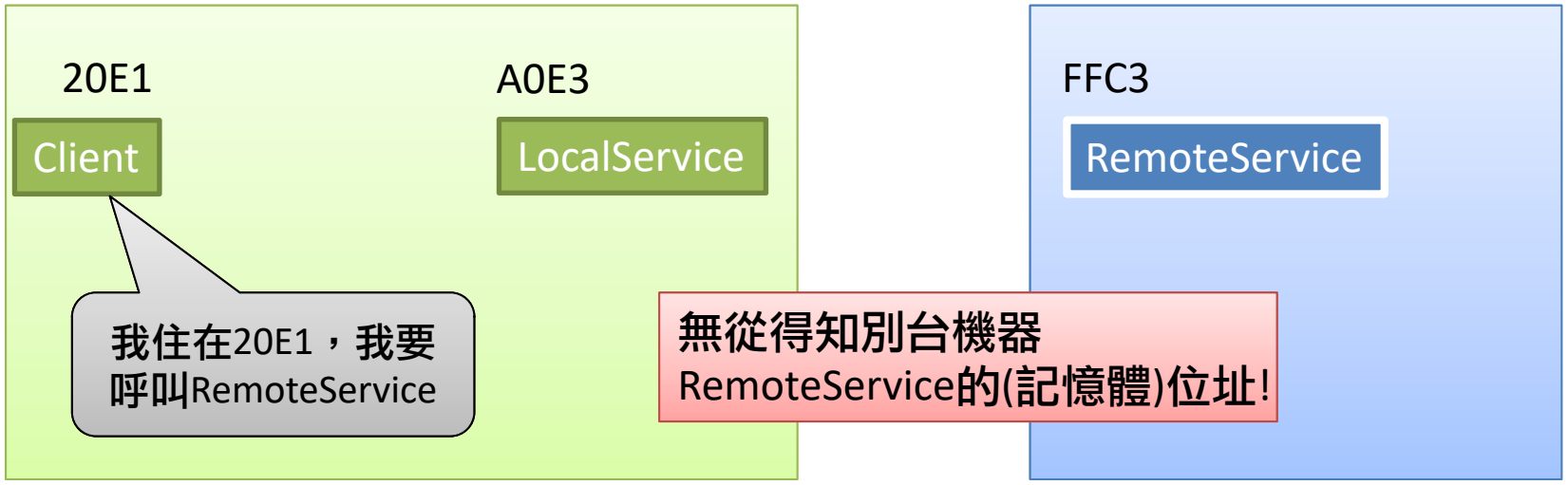
- Motivation
  - A type of the direct communication mechanism
  - Calling procedure on a remote host “as if” calling a local procedure
  - Warning: unaware of “remoting” is considered harmful (Saif and Greaves, 2001)
- Approach
  - Communication between caller & callee can be hidden by using procedure-call mechanism



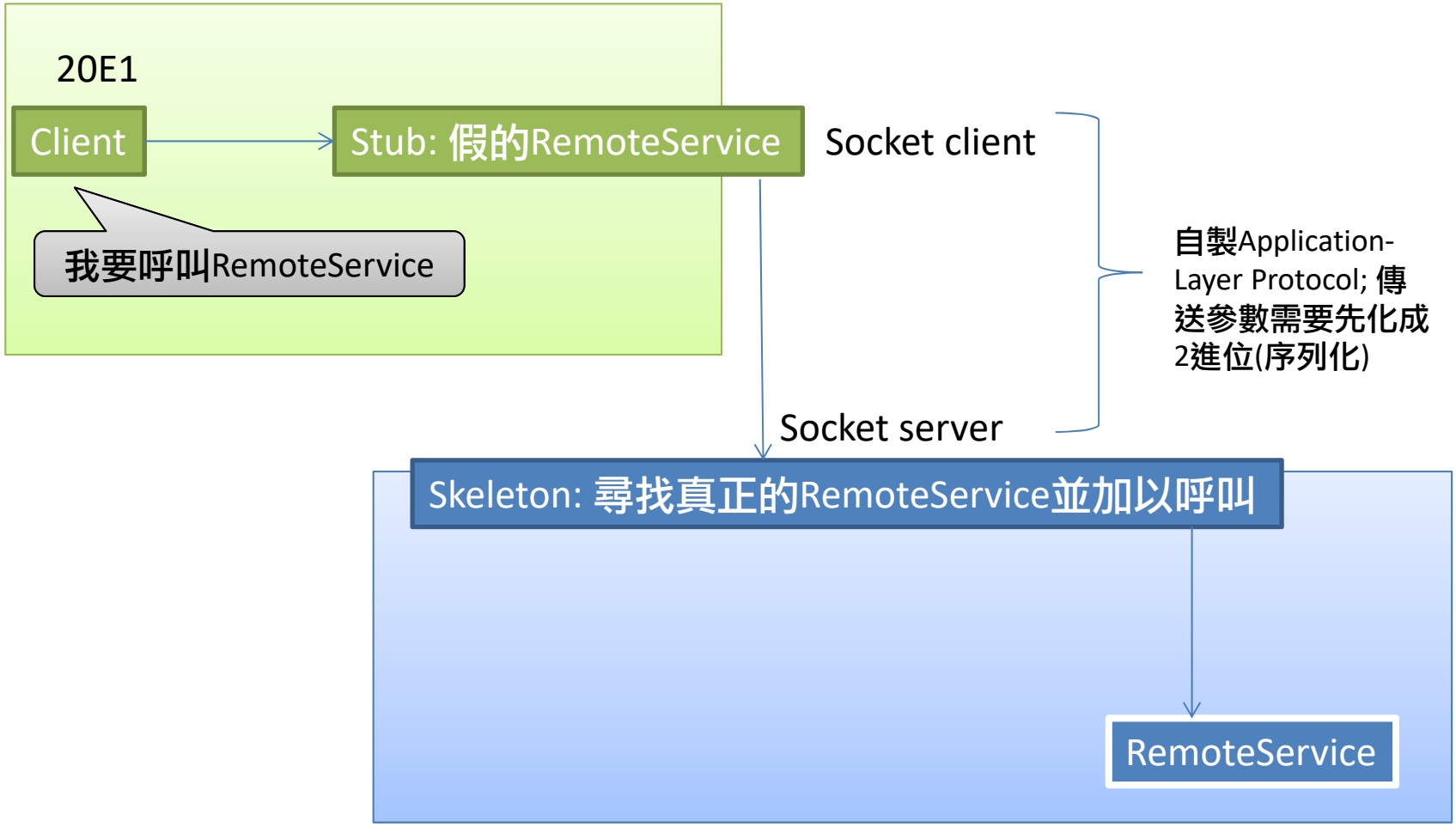

U. Saif and D. J. Greaves, "Communication primitives for ubiquitous systems or RPC considered harmful," Proceedings 21st International Conference on Distributed Computing Systems Workshops, Mesa, AZ, USA, 2001, pp. 240-245, doi: 10.1109/CDCS.2001.918712.
https://ieeexplore.ieee.org/abstract/document/918712

# Remote Procedure Calls



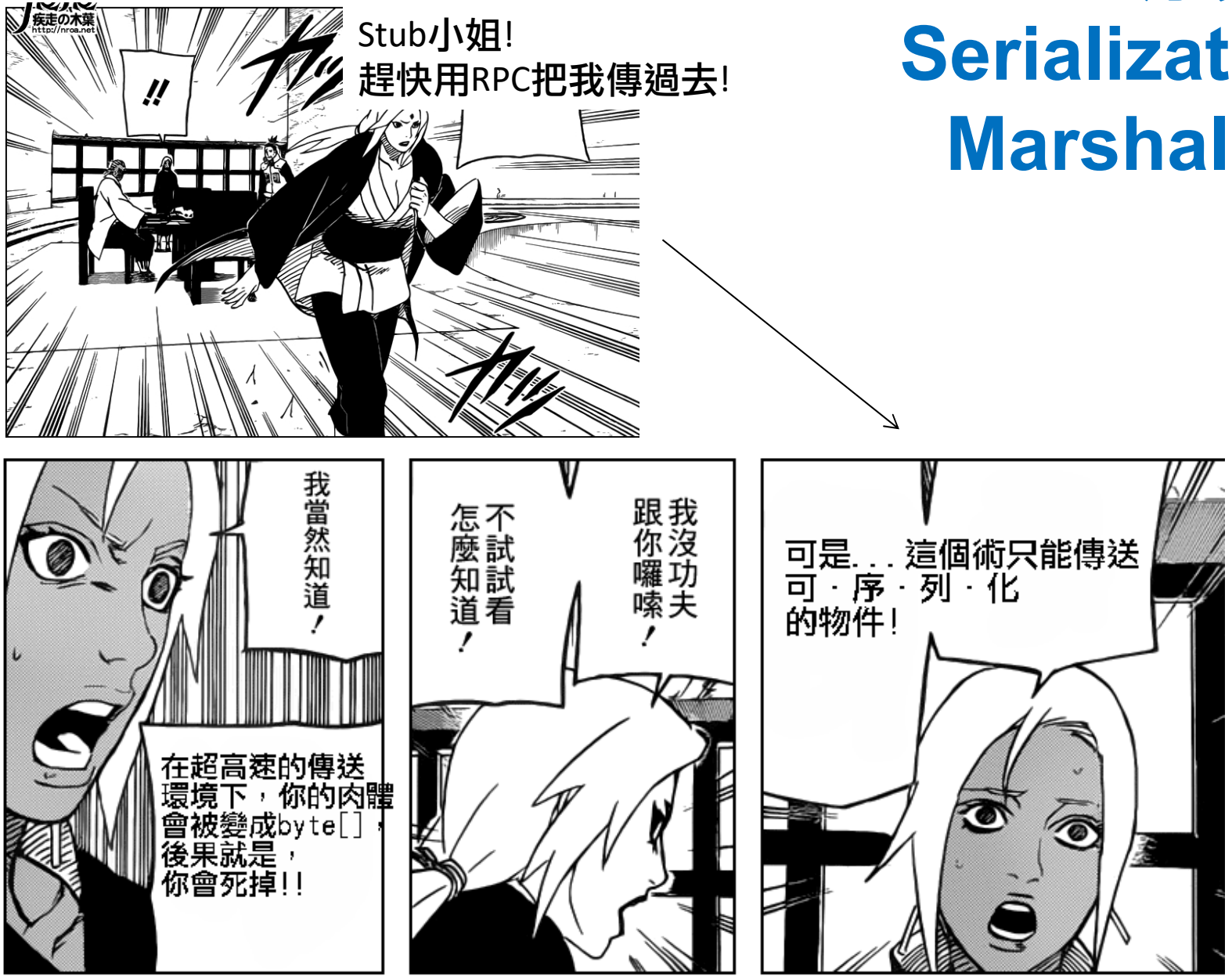
# A Generic Way of Realizing RPC



# Serialization (Marshalling) 序列化

- 參數如何傳送?
  - Client and server machines may have different data representations
    - E.g., byte ordering
  - Serialization序列化: transforming a value into a sequence of bytes
    - Client and server have to agree on the same encoding standard
  - 議題
    - How are basic data values represented (integers, floats, characters)
    - How are complex data values represented (arrays, unions)

# 序列化
# Serialization/ Marshalling

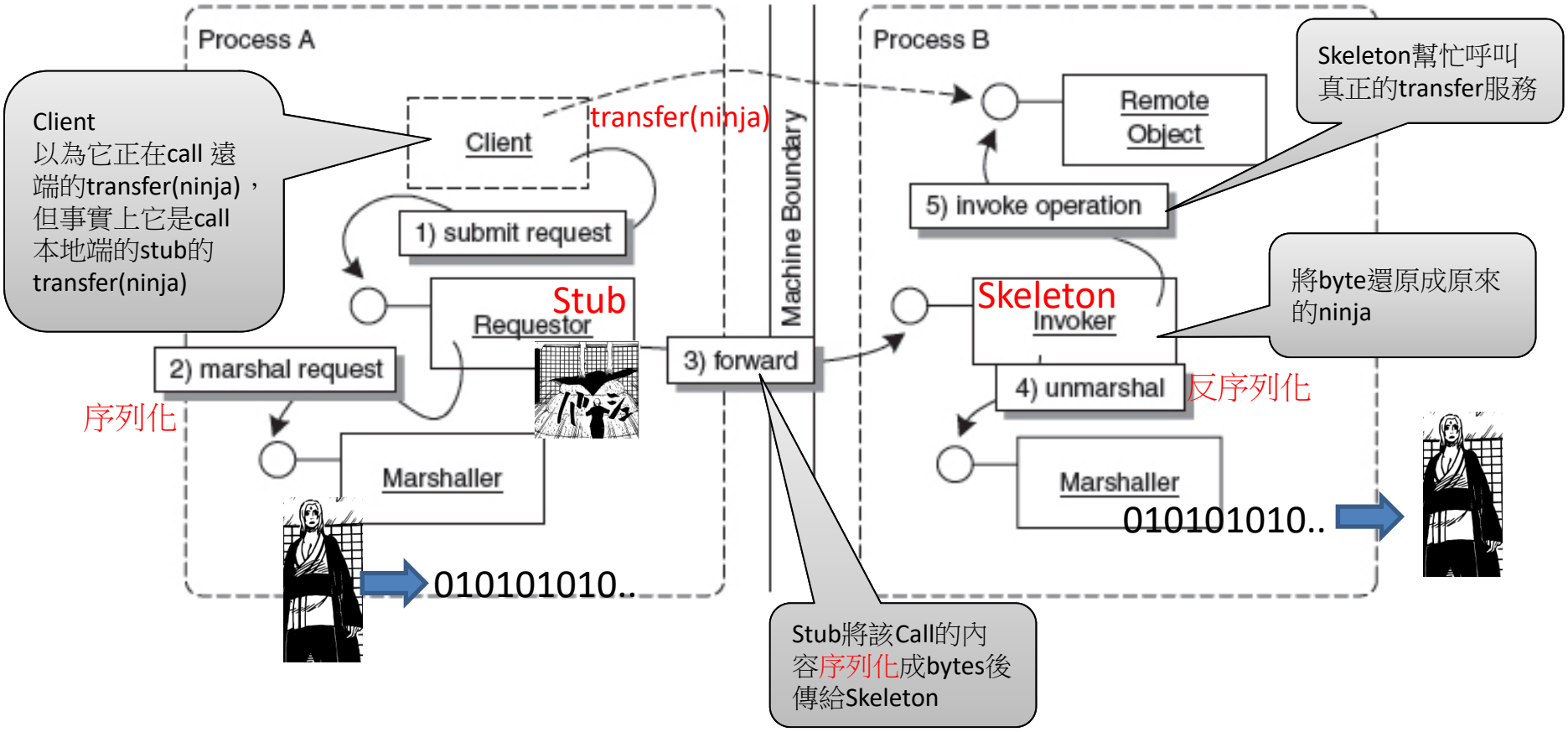



010101010..

Stub

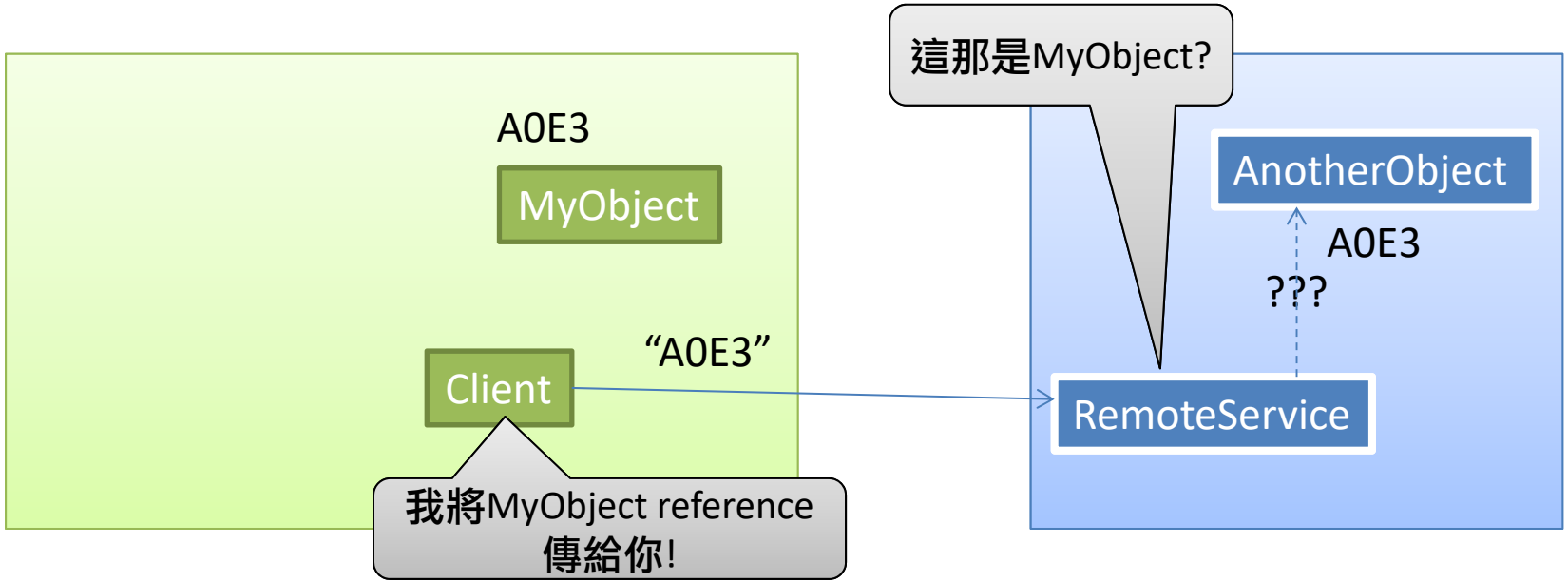
Skeleton

# 呼叫遠端函式(物件)的架構

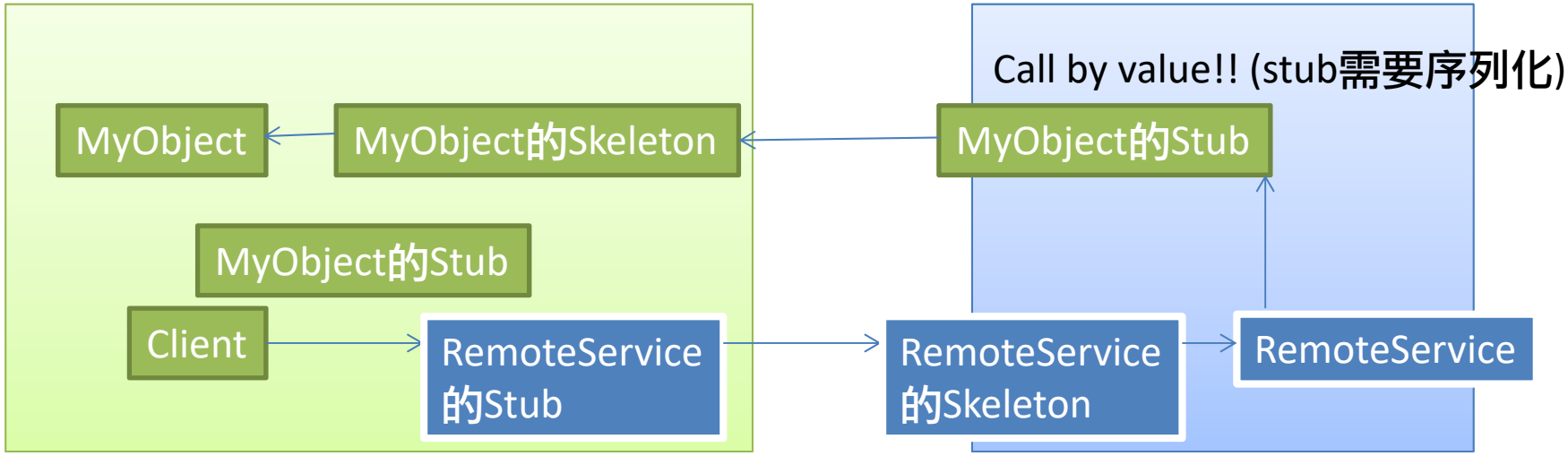


# Passing Reference

- 傳參數的方式
  - Call by reference: 物件
    - 傳物件的記憶體位址
  - Call by value: 基本型別



# 遠端物件參數傳送的真相!!




問題: 如何取得這些Stub?
Stub如何傳送

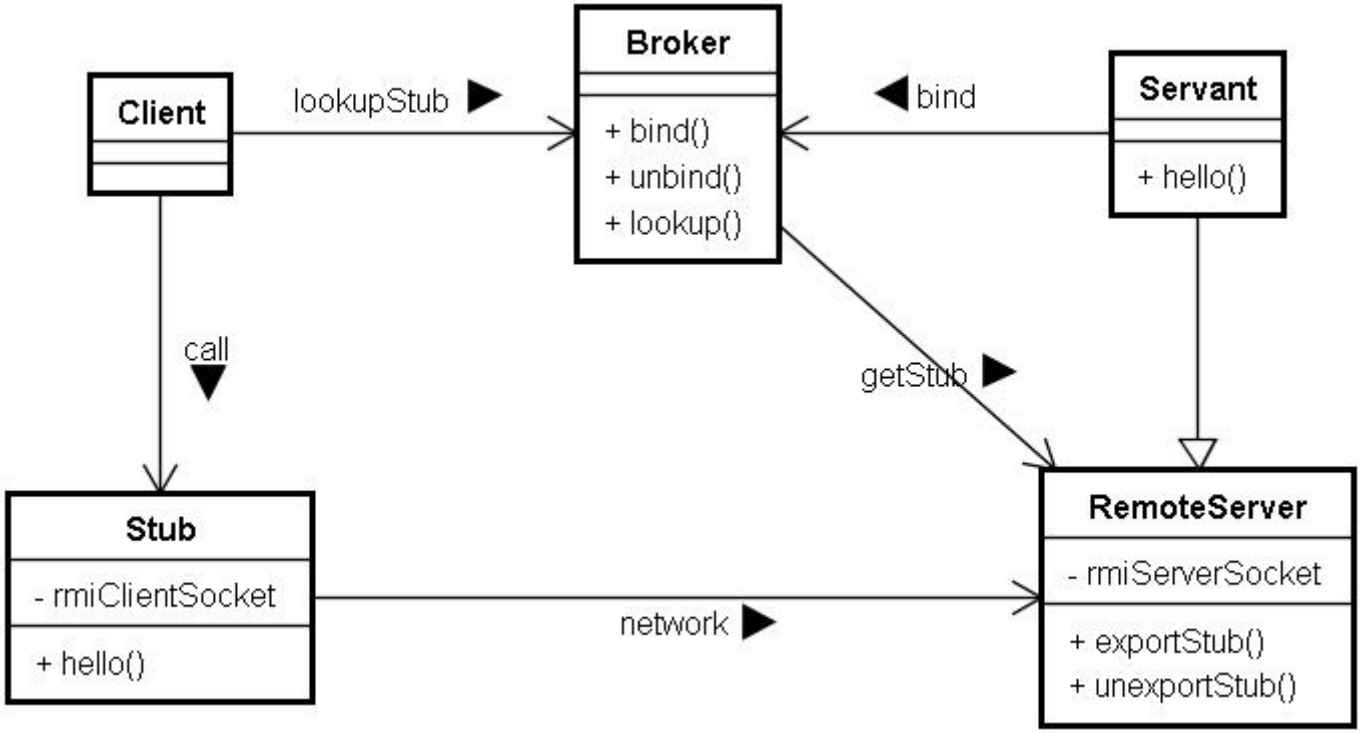
# Broker

- To structure distributed components that interact by remote service invocations
- Forces
  - Location Independence
  - Separation of Concerns
    - Business logic vs. remote communication
  - Transparency

# Structure

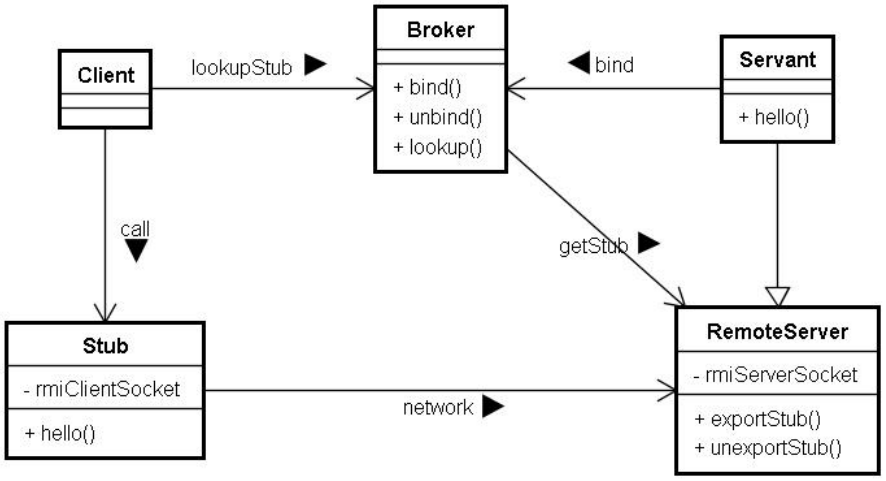
Local call

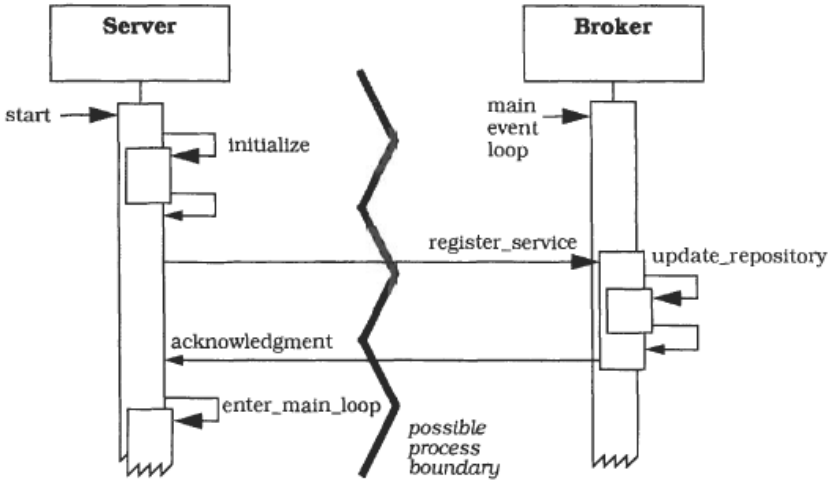

Remote call



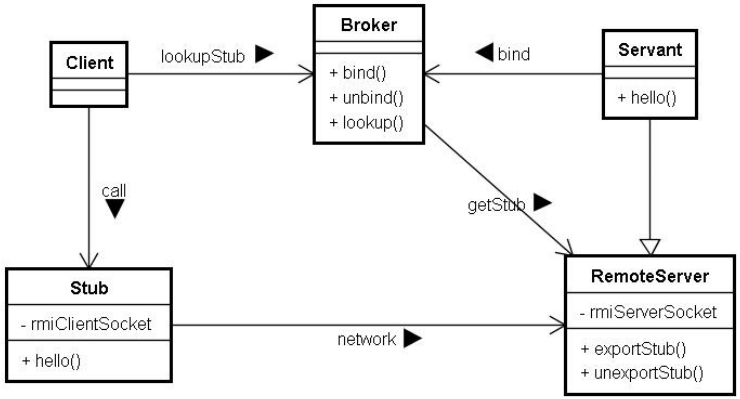
# Broker

- When a new service is added, it registers its information in the broker
- Extract and store the serialized Stub and transfer to calling clients



# Stub




- The Stub allows the hiding of implementation details from the clients such as:
  - The inter-process communication mechanism used for message transfers between client and server
  - The marshaling of parameters and results

# Remote Server

- Receiving requests, unpacking incoming messages, un-marshaling the parameters, and calling the appropriate service.
- Marshaling results and exceptions before sending them to the client

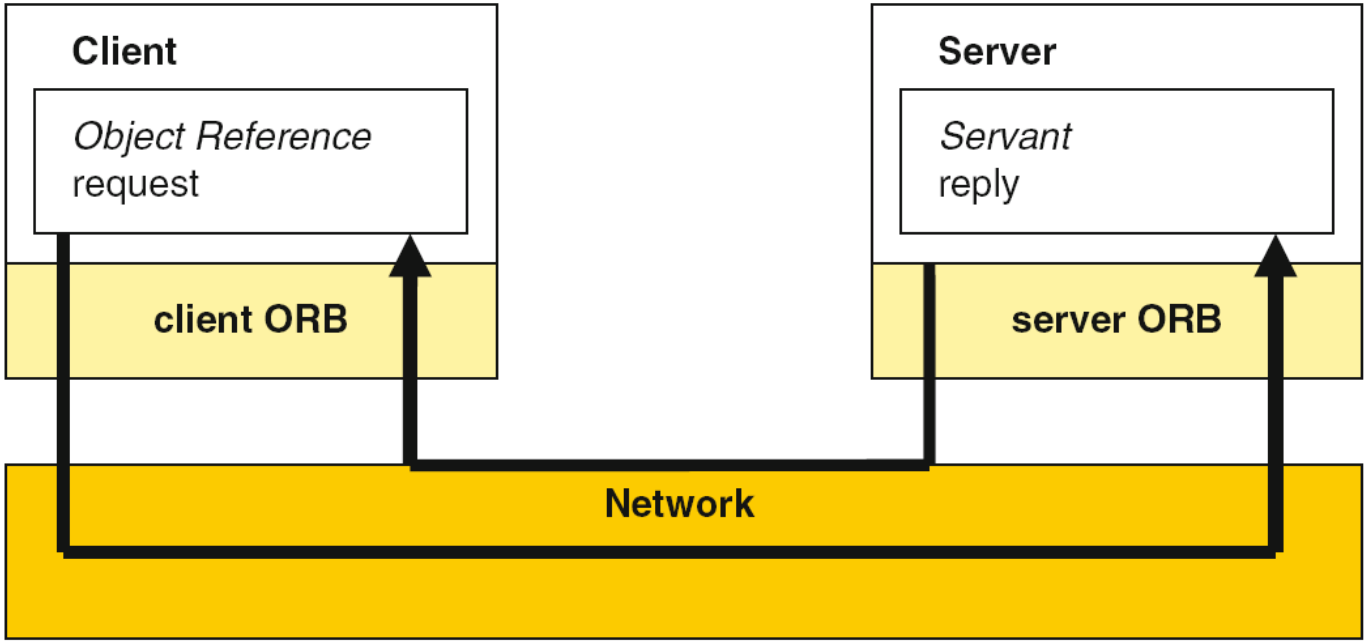
# Known Uses

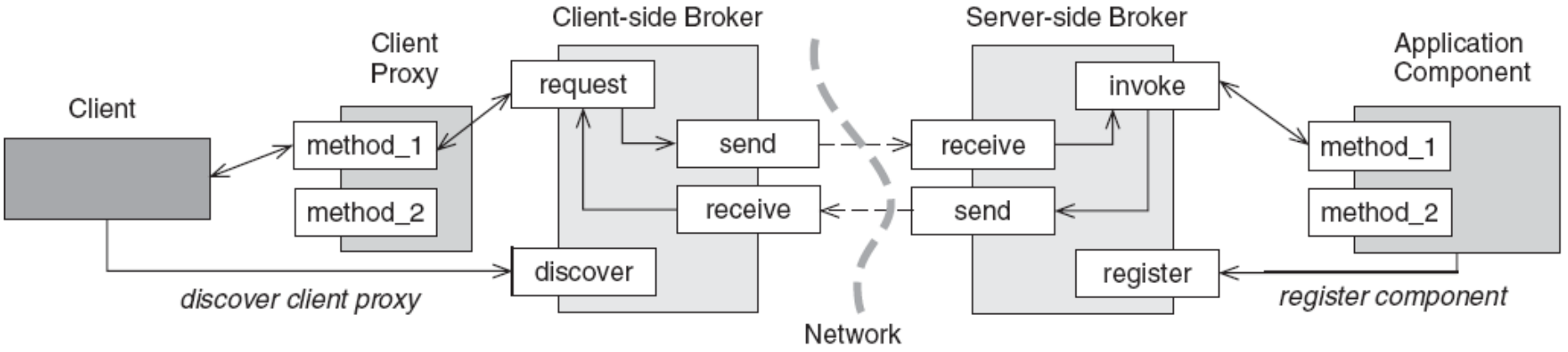
- Best characterized by CORBA, distributed object-based middleware has been in use since the earlier 1990s.
- In CORBA, servant objects support interfaces that are specified using IDL (interface description language)
  - IDL interfaces define the methods that a server object supports, along with the parameter and return types.

```
module ServerExample {
    interface MyObject
    {
        string isAlive();
    };
};
```

```
class MyServant extends _MyObjectImplBase {
  public String isAlive()   {
    return "\nLooks like it…\n";
    }
}
```

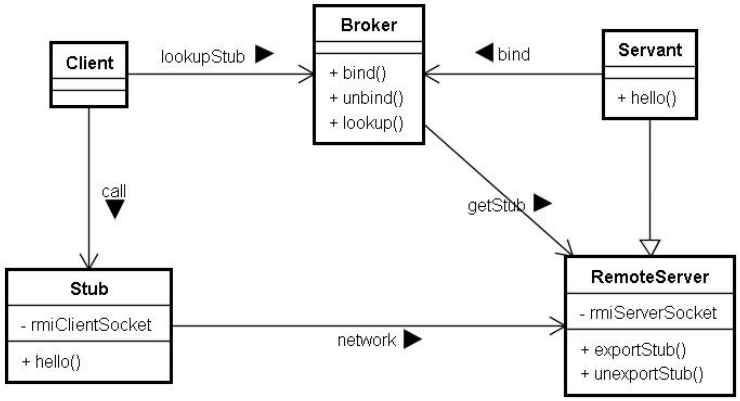
# CORBA's Double-Broker Architecture

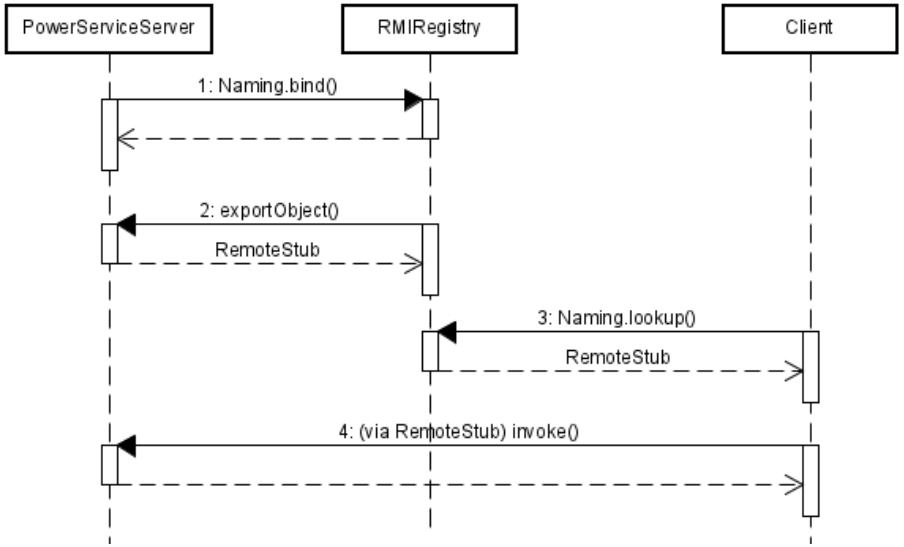


# Known Uses

- Java RMI
  - In contrast to CORBA the servant interfaces are not written in an abstract IDL, but in Java. Consequently RMI is limited to the usage of Java.
  - Stub has to be generated using RMIC (before JDK 5)
  - A broker service (so called RMI registry) allows clients to look up servant identifiers



# Known Uses

- .NET Remoting (deprecated)
  - Stub is created dynamically at runtime
  - The interface description can be provided by MSIL-Code or by WSDL
  - .NET Remoting doesn't have a central broker component. Clients have to know the object reference of the servant in advance

```
TcpChannel channel = new TcpChannel(8080);
ChannelServices.RegisterChannel(channel, ensureSecurity : true);
RemotingConfiguration.RegisterWellKnownServiceType(
   typeof(RemotingServer),
   "RemotingServer",
   WellKnownObjectMode.Singleton);
Console.WriteLine("RemotingServer is running.  Press ENTER to terminate...");
Console.ReadLine();
```

```
public class RemotingServer : MarshalByRefObject
{
   public Customer GetCustomer(int customerId) { … }
}
```

# Known Uses

- Windows Communication Foundation (WCF)

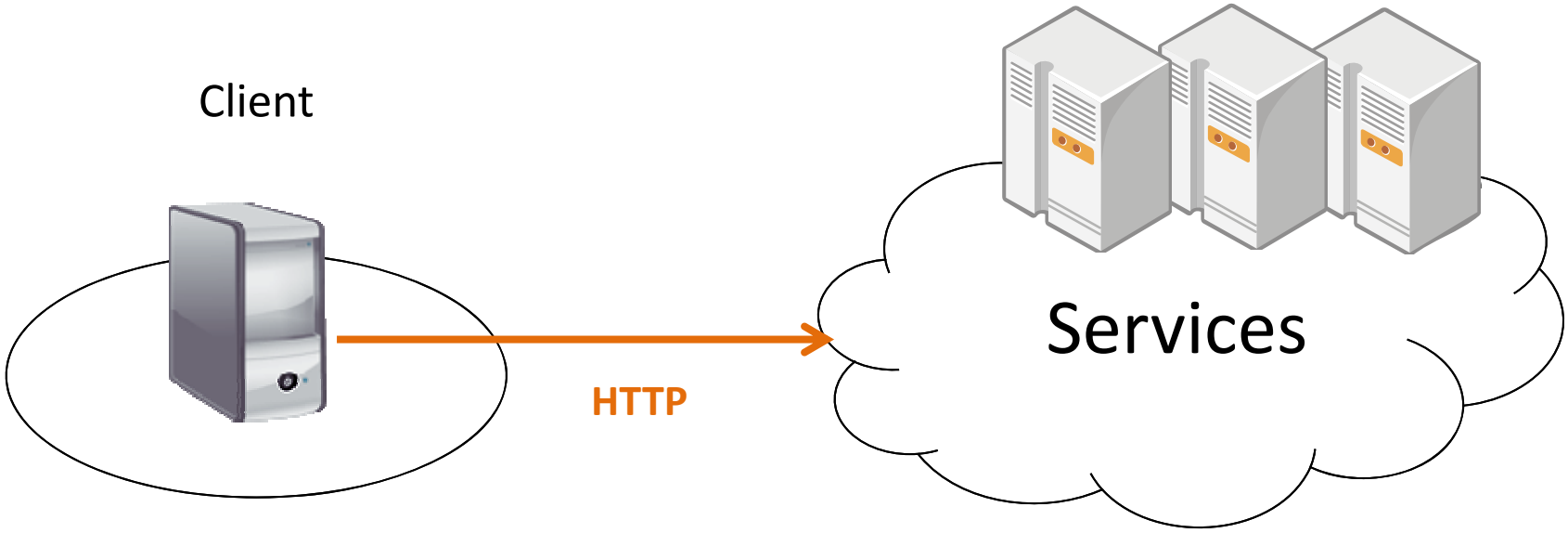
```
NetTcpBinding binding = new NetTcpBinding();
Uri baseAddress = new Uri("net.tcp://localhost:8000/wcfserver");

using (ServiceHost serviceHost = new ServiceHost(typeof(WCFServer), baseAddress))
{
   serviceHost.AddServiceEndpoint(typeof(IWCFServer), binding, baseAddress);
   serviceHost.Open();

   Console.WriteLine($"The WCF server is ready at {baseAddress}.");
   Console.WriteLine("Press <ENTER> to terminate service...");
   Console.WriteLine();
   Console.ReadLine();
}
```

```
[ServiceContract]
public interface IWCFServer
{
   [OperationContract]
   Customer GetCustomer(int customerId);
}
```

```
public class WCFServer : IWCFServer
{
   public Customer GetCustomer(int
customerId) { … }
}
```

# Case: Web Service

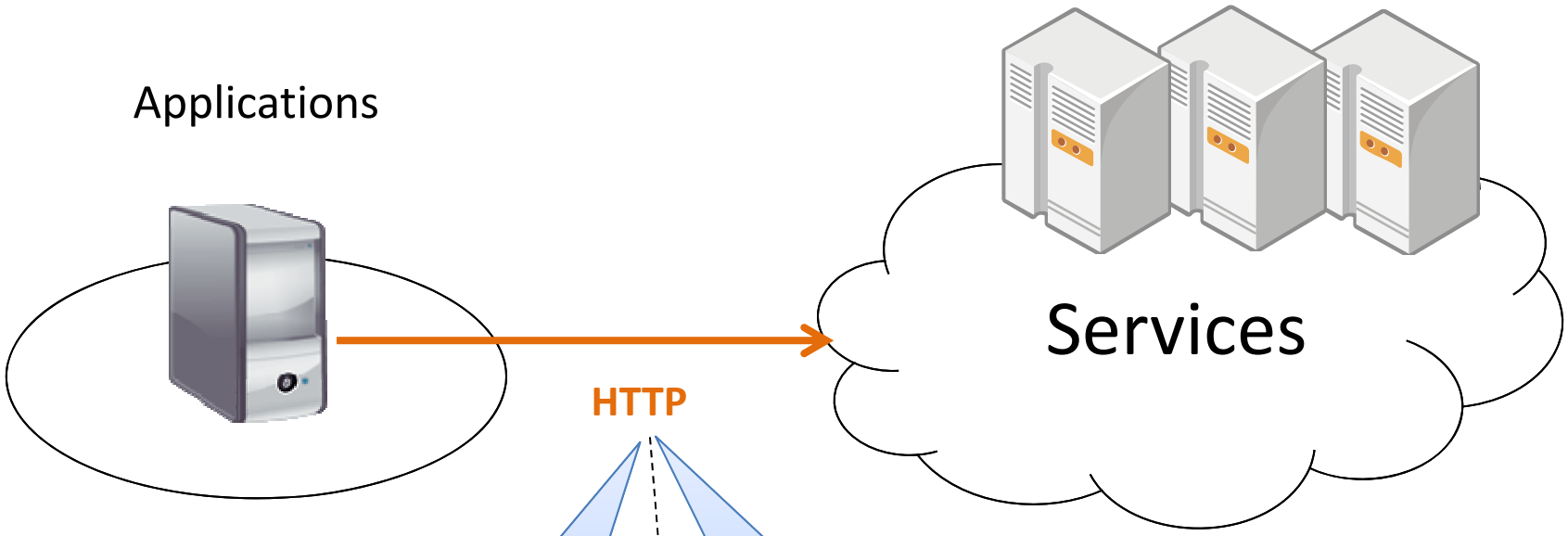
- 可透過Internet (HTTP)存取的遠端業務邏輯
  - Web Services是實現SOA的主要技術



# Web Services: 二種主要實現方式

- SOAP
  - 將服務視為遠端函式
  - 依照一定格式將呼叫/回應以XML編碼 (SOAP, Simple Object Access Protocol)
  - 只將HTTP拿來做為訊息運送工具
  - 使用WSDL(Web Service Description Language)描述服務內容
- RESTful
  - 將HTTP視為應用程式平台，將服務視為物件(資源)
  - 使用HTTP方法(GET/POST/…)操作資源
  - 不限定訊息格式 (XML, JSON或其它)
  - 有多種方式可用來描述服務
    - Swagger (Open API)
    - WADL

# Web Services



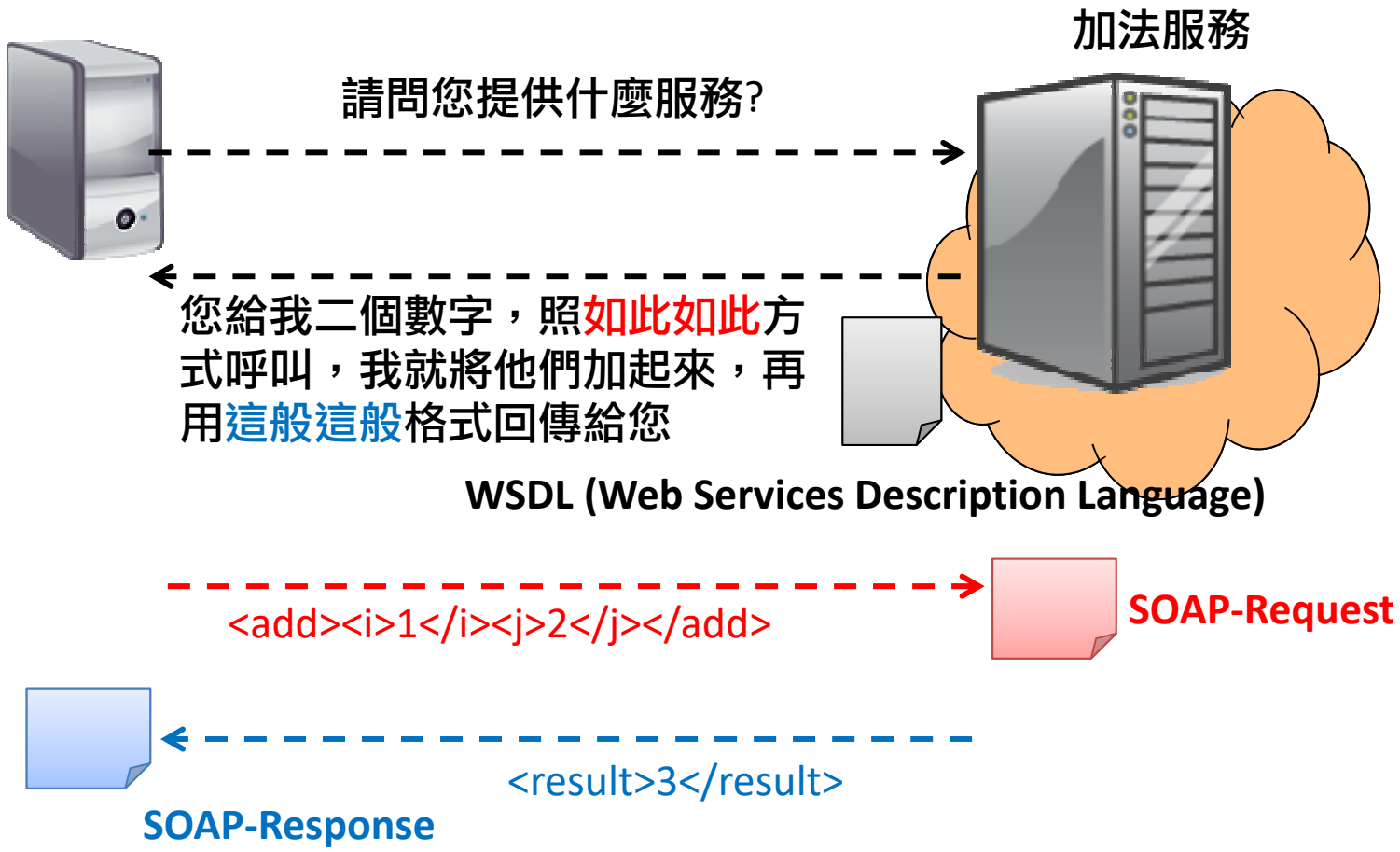

在雲端環境中，主[illegible]之間透過[illegible]et標準協定進行呼叫

用什麼方式呼叫（存取方式）?

有那些服務可呼叫（服務內容）?

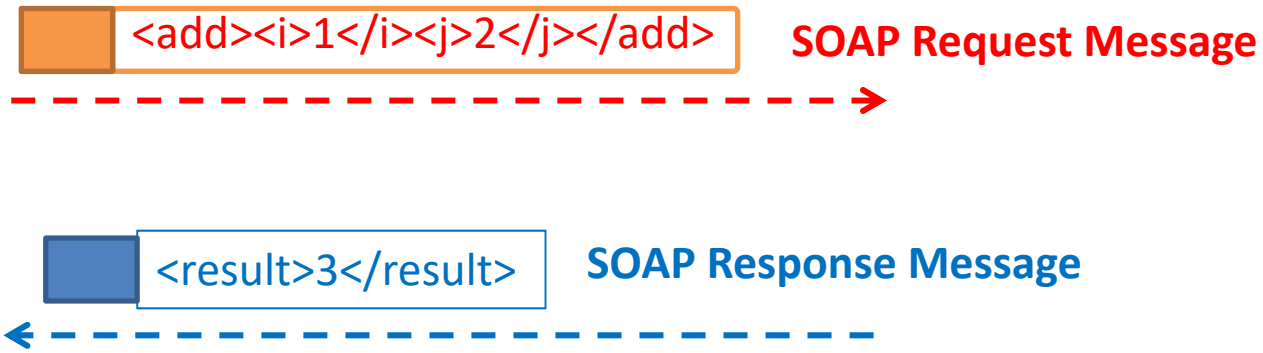
# WSDL 與SOAP

- 描述所提供存取方式及服務內容的標準規範



# SOAP

- Simple Object Access Protocol
- An XML-based communication protocol
  - let applications exchange information over HTTP




(在TCP/IP層，　　網路傳送訊息單元稱為Packet
在Application層，網路傳送訊息單元稱為Message)

# SOAP

- What's in the SOAP?
  - A packaging model
  - A serialization mechanism
  - An RPC mechanism

# SOAP 封包結構


```
<soap:Envelope>
  <soap:Header/>
  <soap:Body>
    <add>
      <x>1</x>
      <y>1</y>
    </add>
  </soap:Body>
</soap:Envelope>
```

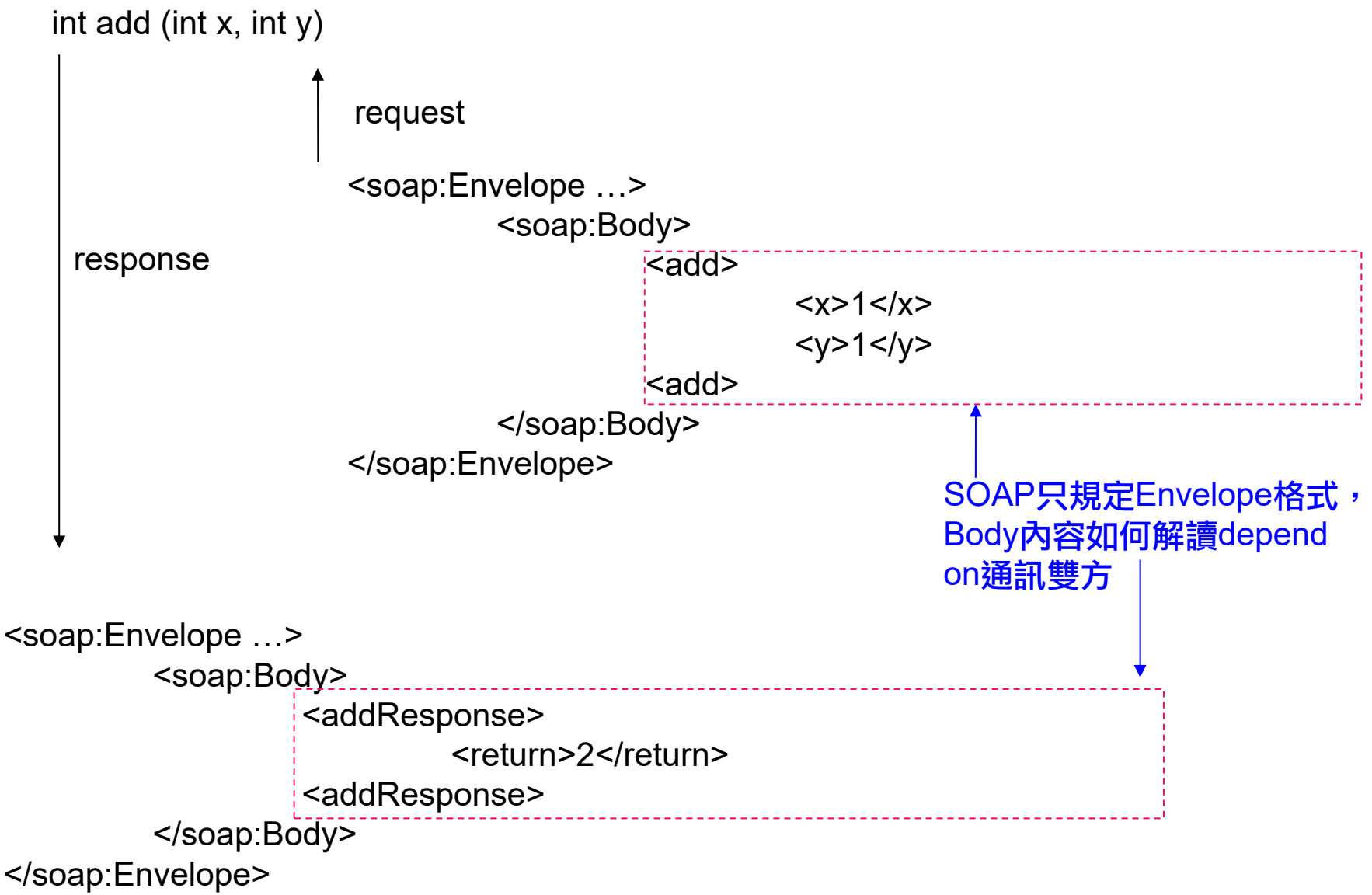
```
<soap:Envelope>
  <soap:Header/>
  <soap:Body>
    <addResponse>
      <return>2</return>
    </addResponse>
  </soap:Body>
</soap:Envelope>
```

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

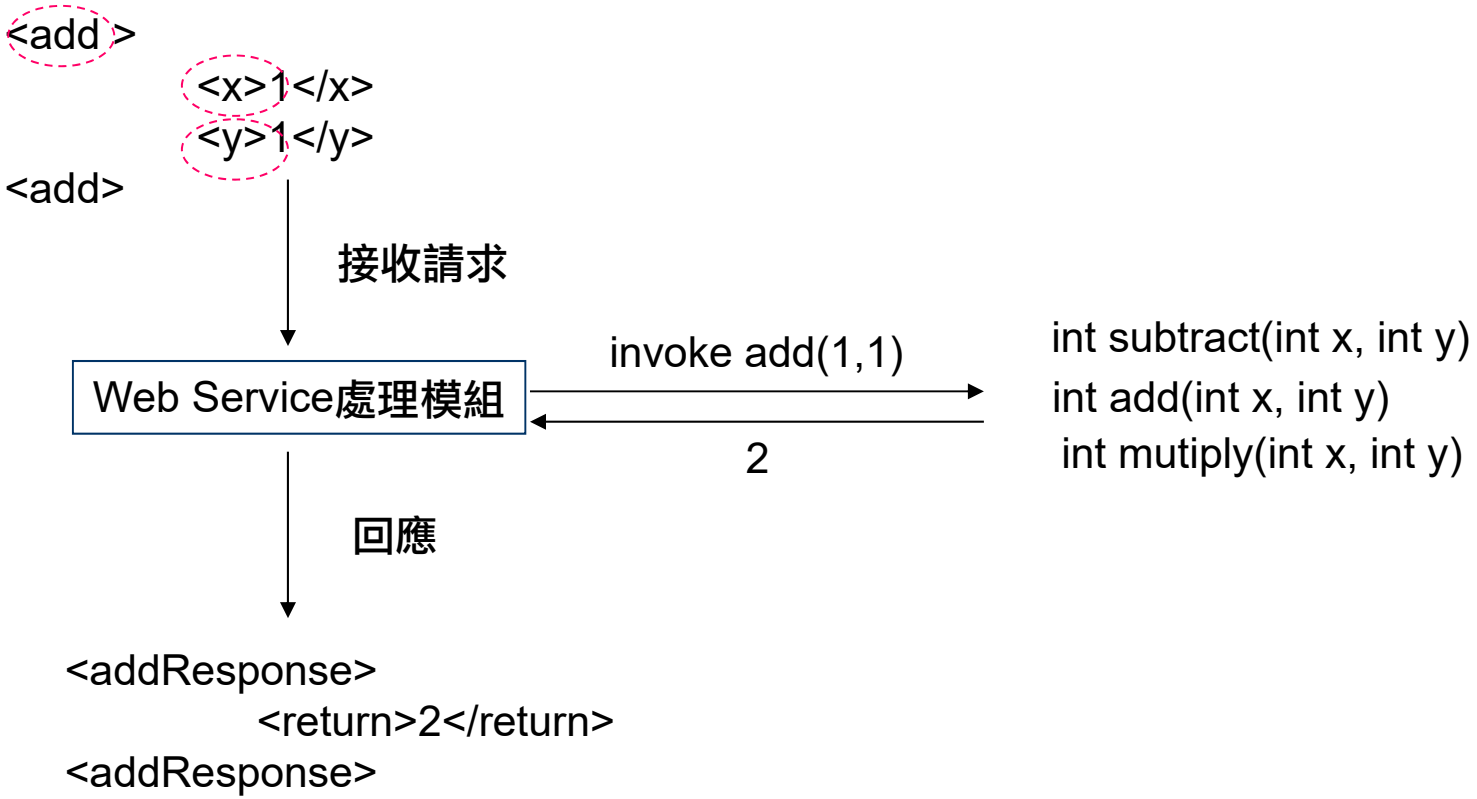
# Demo

- SOAP server and client
  - Simple adder service

# 使用SOAP進行RPC



# SOAP Server




處理模組怎麼知道add一定是對應到method name, x, y是argument?

# WSDL

- 雙方如何約定、解讀 SOAP Body內容?
  - 對照http://localhost:8192/Adder?wsdl與SOAP Messages的內容

```
<wsdl:message name="add">
      <wsdl:part name="x" type="xsd:int"> </wsdl:part>
      <wsdl:part name="y" type="xsd:int"> </wsdl:part>
</wsdl:message>
<wsdl:message name="addResponse">
      <wsdl:part name="return" type="xsd:int"> </wsdl:part>
</wsdl:message>

<wsdl:portType name="Calculator">
     <wsdl:operation name="add">
        <wsdl:input message="add" name="add"> </wsdl:input>
        <wsdl:output message="addResponse" name="addResponse"> </wsdl:output>
     </wsdl:operation>
</wsdl:portType>
```

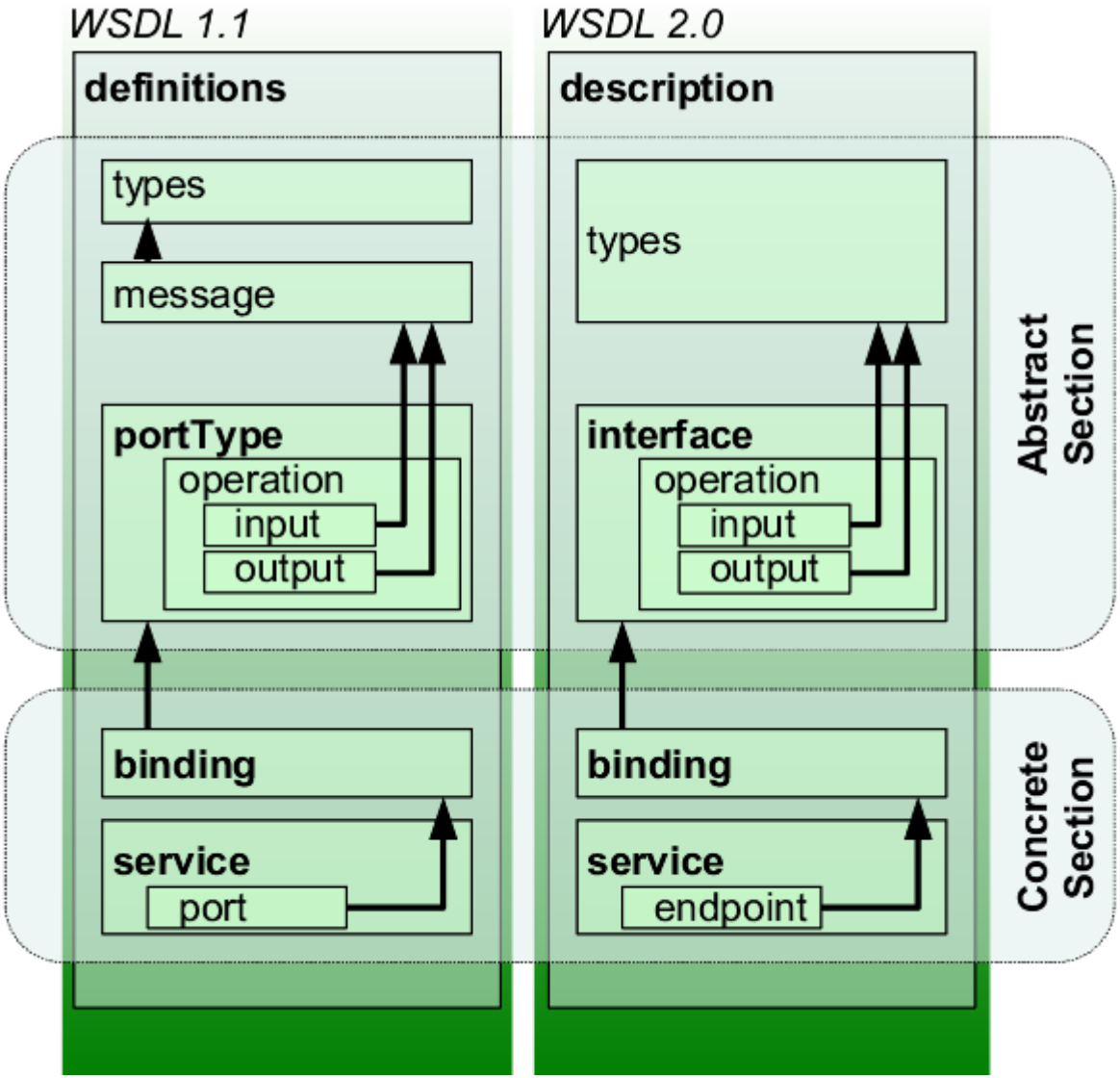
SOAP範例

```
<add >
   <x>1</x>
   <y>1</y>
<add>
```

SOAP範例

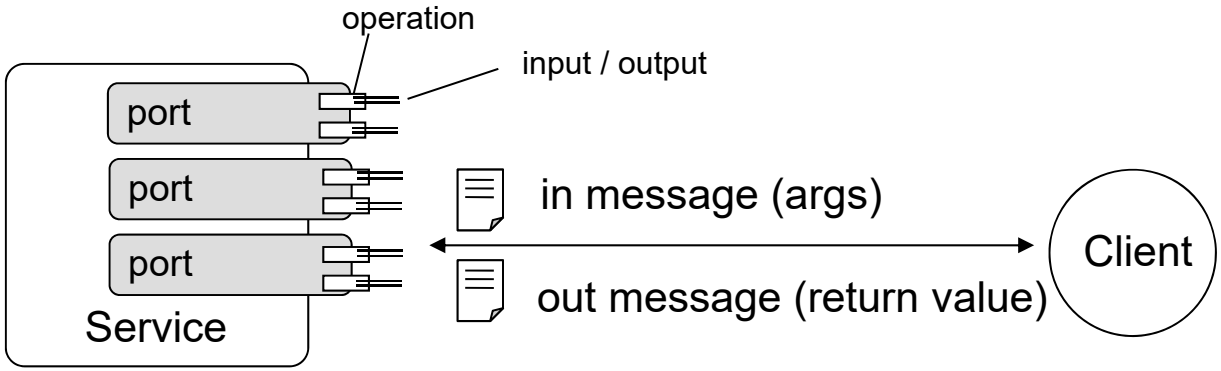
```
<addResponse>
    <return>2</return>
<addResponse>
```

# WSDL 結構

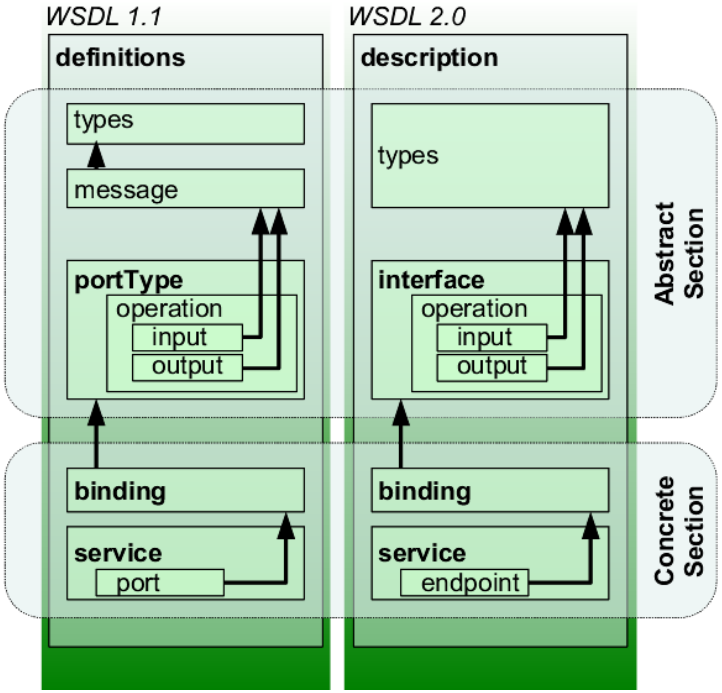



(看一個實際的範例)

# WSDL 結構 (2)



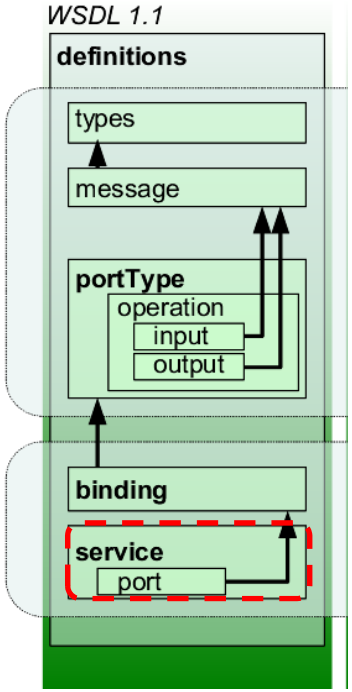

In / out messages are constrainted by XSDs
Ports are constrainted by portTypes



# Service and Port

- Service
  - Represent a web service
  - Consists of a collection of “ports”
- Port
  - Location of a service (as URL)
  - Bind to an “wsdl:binding” element

```
<wsdl:service name="CalculatorImplService">
   <wsdl:port name="CalculatorImplPort“
                  binding="tns:CalculatorImplServiceSoapBinding" >
      <soap:address location="http://localhost:8192/Adder"/>
   </wsdl:port>
</wsdl:service>
```



# Binding

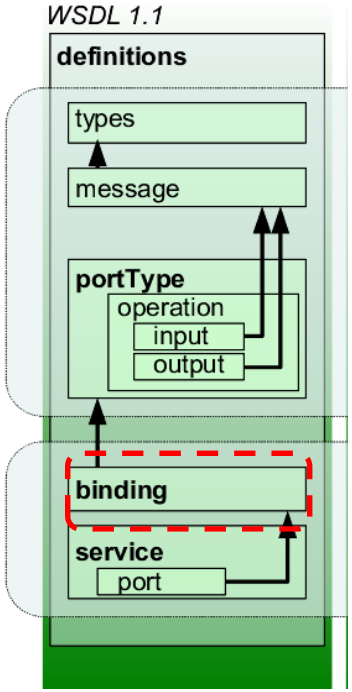
指向PortType (Interface)

```
<wsdl:binding name="CalculatorImplServiceSoapBinding"   type="tns:Calculator">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="add"> 指定每個operation的binding方式
        <soap:operation soapAction=“”  style="document"/>
        <wsdl:input name="add"><soap:body use="literal"/></wsdl:input>
        <wsdl:output name="addResponse"><soap:body use="literal"/></wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

RPC 只有參數型別用XSD規範
Document 整份文件都用XSD規範

- 功能
  - 選擇用什麼協定: SOAP, HTTP, MIME,…
  - 選擇用什麼格式傳送呼叫: SOAP RPC or XSD Document
  - 參數型別是否要隨訊息傳送: encoded (yes)  or literal (no)

Encoded: 將型別資訊放在每個SOAP封包中傳送 (type encoded)
Literal: 反之;
目前大部份選用literal，因為型別資訊應該在第一次呼叫，取得wsdl中獲得就好了，不用每次傳送

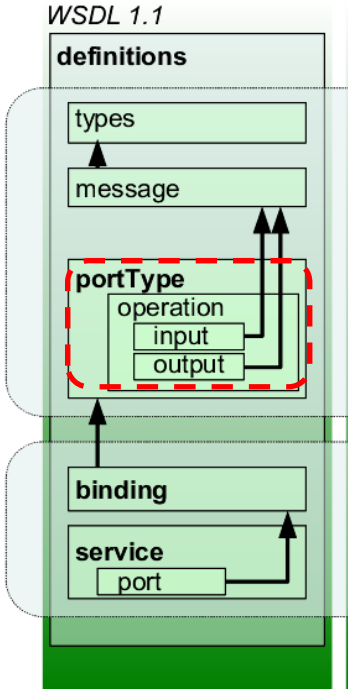


# PortType (Interface)

- 功能
  - 宣告此一Interface中有那些operations
  - 宣告這些operations中有那些參數/傳回值
  - 參數型別定義在message/types區段中

```
<wsdl:portType name="Calculator">
    <wsdl:operation name="add">
        <wsdl:input message="tns:add" name="add“ />
        <wsdl:output message="tns:addResponse" name="addResponse“ />
    </wsdl:operation>
</wsdl:portType>
```

指向Message

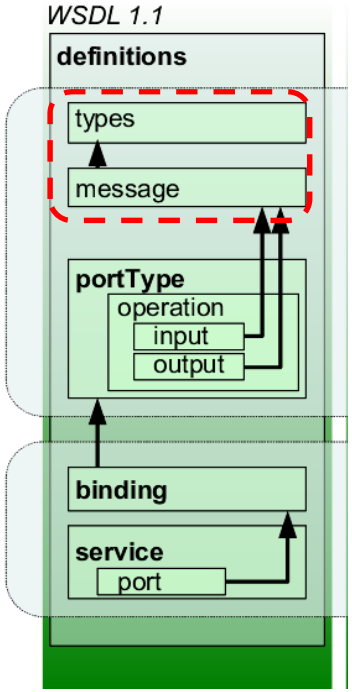


# Message / Types

- 功能
  - 定義所要傳送的訊息(以呼叫來說，則是用來定義方法簽章或參數型別)

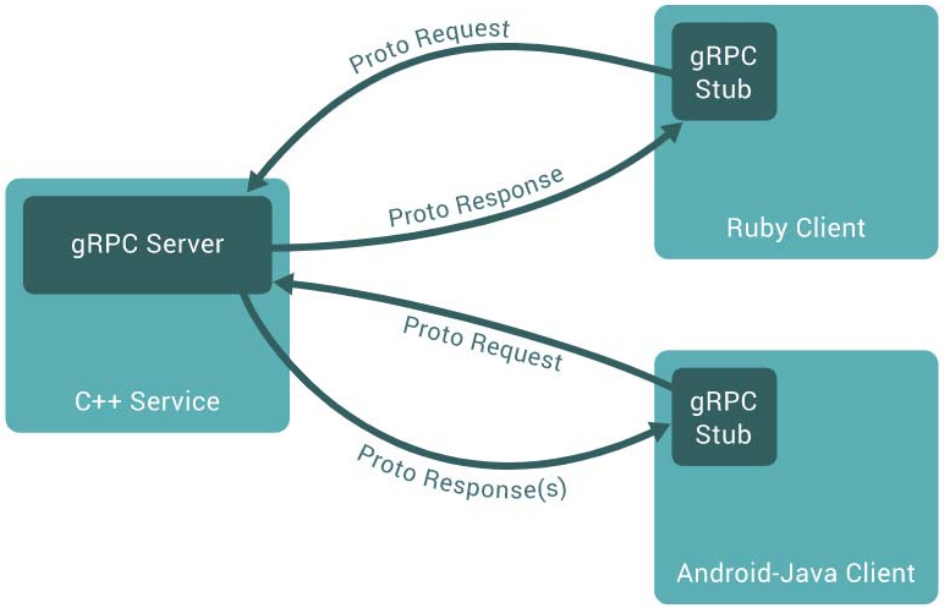
```
<wsdl:message name="add">
   <wsdl:part element="tns:add" name="parameters“ />
</wsdl:message>
<wsdl:message name="addResponse">
   <wsdl:part element="tns:addResponse" name="parameters“ />
</wsdl:message>

 <wsdl:types>
    <xsd:schema targetNamespace="http://lab1.soa.nccu/">
      <xsd:element name="add"  type="xsd:int"/>
      <xsd:element name="addResponse"  type="xsd:int"/>
    </xsd:schema>
 </wsdl:types>
```

指向types



# Case: gRPC

- A modern open source high performance RPC framework that can run in any environment
  - Efficiently connect services in and across data centers
  - Technology stack
    - HTTP 2
    - Protocol Buffers



# Case: gRPC

- Stubby
  - The general RPC framework used by Google internally
- gRPC: standardized and general-purpose Stubby
  - Related by Google at 2015 and joined CNCF
  - Binary protocol (比較: text protocol – REST and SOAP)
  - Strong typed (must define IDL)
  - Built-in features
    - Authentication
    - Encryption
    - Deadline/timeouts
    - Metadata/service discovery
    - Compression
    - Load-balance

# Case: gRPC

- 何時不合適
  - External facing services
    - 普及度、client可選擇性與彈性
  - 經常需要修改interfaces
    - Client/server都要重新改code

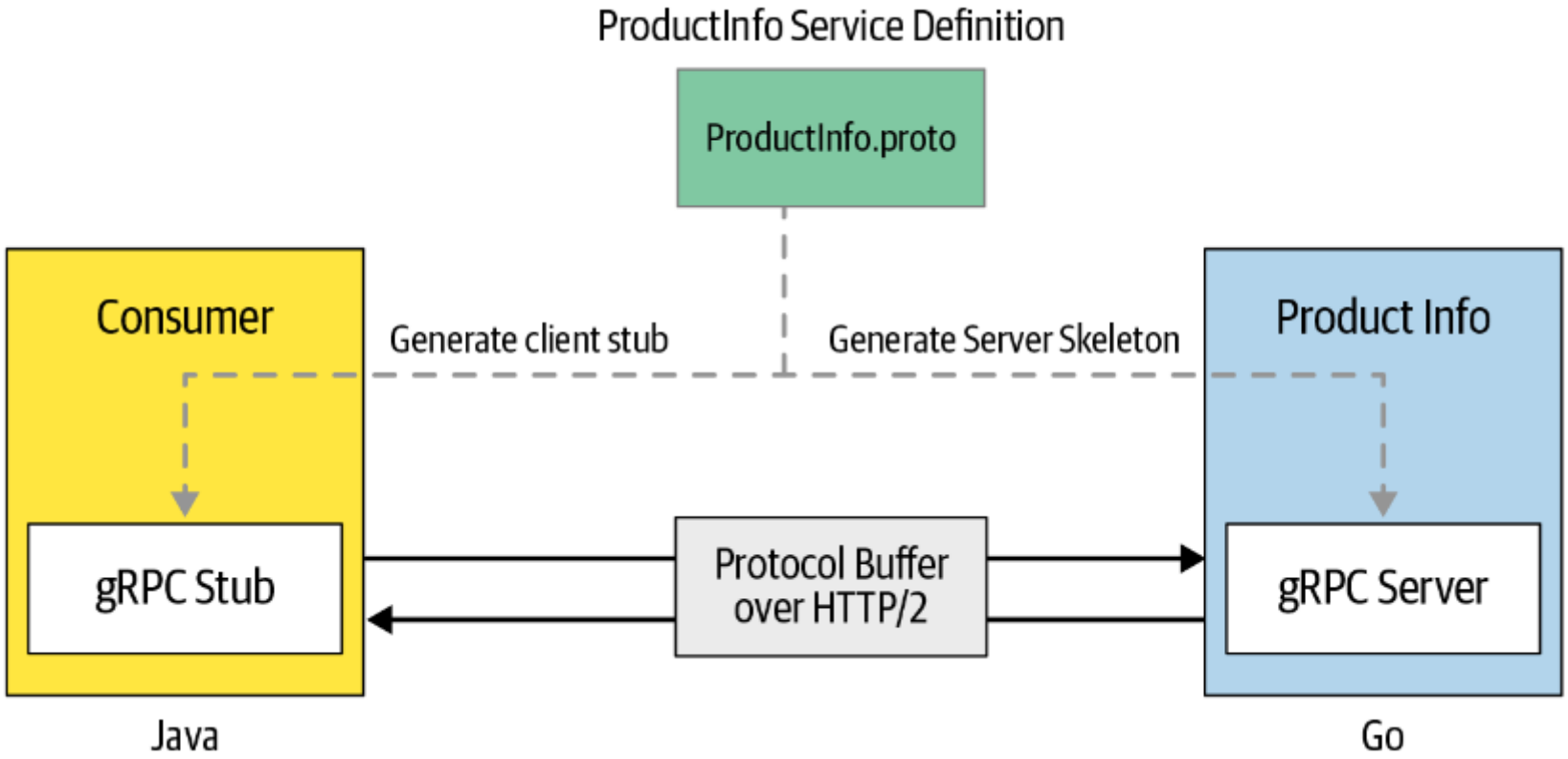
# gRPC in the Realworld

- gRPC has been widely adopted for building microservices and cloud native applications
- Netflix
  - Initially using in-house RESTful solution on HTTP/1.1
  - With the adoption of gRPC, Netflix has seen a massive boost in developer productivity
    - Creating a client, which could take up to two to three weeks, takes a matter of minutes with gRPC.

# gRPC in the Realworld

- Dropbox
  - Dropbox runs hundreds of polyglot microservices, which exchange millions of requests per second
  - Initial solution
    - A homegrown RPC framework with a custom protocol for manual serialization
    - Apache Thrift
    - Legacy HTTP/1.1-based RPC framework + protobuf
  - New solution: Courier
    - A gRPC-based RPC framework
    - A customized solution to meet specific requirements like authentication, authorization, service discovery, service statistics, event logging, and tracing tools

# A Server and a Client based on gRPC



```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

# Demo

# Server程式碼

Input
(a structure)

Output
(a function)

sendUnaryData(error, value [, trailer] [, flags])

```
function sayHello(call, callback) {
  callback(null, {message: 'Hello ' + call.request.name});
}

function sayHelloAgain(call, callback) {
  callback(null, {message: 'Hello again, ' + call.request.name});
}

function main() {
  var server = new grpc.Server();
  server.addService(hello_proto.Greeter.service,
                    {sayHello: sayHello, sayHelloAgain: sayHelloAgain});
  server.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());
  server.start();
}
```

https://grpc.github.io/grpc/node/grpc.Server.html

# Client端程式碼

傳入參數　　接收回傳，並處理之

```
function main() {

  var client = new hello_proto.Greeter('localhost:50051',
                                       grpc.credentials.createInsecure());

  client.sayHello({name: 'you'}, function(err, response) {
    console.log('Greeting:', response.message);
  });

   client.sayHelloAgain({name: 'you'}, function(err, response) {
    console.log('Greeting:', response.message);
  });

}
```

# Q & A